ESD-TR-68-143, Vol. I

EVOLUTIONARY SYSTEM FOR DATA PROCESSING
SYSTEM DESCRIPTION

Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

January 1968

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0254 by Center for Exploratory Studies, International Business Machines Corporation, Rockville, Maryland.)

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

FSC 68-0688/3053

ESD-TR-68-143, Vol. I

EVOLUTIONARY SYSTEM FOR DATA PROCESSING
SYSTEM DESCRIPTION

Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

January 1968

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0254 by Center for Exploratory Studies, International Business Machines Corporation, Rockville, Maryland.)

FOREWORD

This report presents the results of a study of the specifications for an information system intended to support the design, production and maintenance of large computer programming systems. Called Evolutionary System for Data Processing, or ESDP, it was begun as an internal IBM project in 1965 by the Center for Exploratory Studies of the Federal Systems Division and continued under Air Force sponsorship during 1967 and early 1968.

This work has been performed under contract number F19628-67-C0354 for the Electronic Systems Division, U.S. Air Force Systems Command. The project monitor was Mr. John Goodenough, ESLFE.

The authors wish to express their appreciation for the encouragement and assistance provided by Dr. John Egan, formerly of ESD, and their colleagues Dr. Harlan D. Mills and Mr. Michael Dyer.

This report is in four volumes: Volume 1, System Description; Volume 2, Control and Use of the System; Volume 3, The CAINT Executive Language and Instruction Generator; and Volume 4, Programming Specifications. This report was submitted on January 31, 1968.

This report has been reviewed and is approved.

Sylvia R. Mayer

SYLVIA R. MAYER
Project Officer

William F. Heisler

WILLIAM F. HEISLER, Col, USAF
Chief, Command Systems Division

ABSTRACT

ESDP is a proposed system whose purpose is to acquire, store, retrieve, publish and disseminate all documentation, exclusive of graphics, concerned with a large computer programming activity. Documentation is deemed to consist, not only of final or formally published after-the-fact reports, but of working files, design and change notices, informal drafts, management reports--in fact, the entire recordable rationale underlying a programming system. Maximum attention has been concentrated on the means of acquiring and organizing documentation. Two major, complementary approaches are proposed. The first is called Program Analysis and is a process of extracting documentation directly from completed programs. The second is called Computer Assisted Interrogation and is a process of eliciting information directly from the programmer, through on-line communication terminals. The former provides canonical data about the program's structure. The latter provides explanatory material about all aspects of the program, and in the absence of canonical data, may provide tentative structural information as well. The conclusion of the study group is that ESDP is a feasible concept with present-day technology and that it will materially benefit using organizations in the production of programs and in guiding their evolution as requirements change. Its value will be greater for larger organizations, whose internal communications difficulties tend to cause truly gigantic inefficiencies. Its implementation as a support system for such projects would require a significant quantum of investment in order to produce these benefits and is predicated on the use of a computer system dedicated solely to the use of ESDP.

Volume I

System Definition

ILLUSTRATIONS

TABLES

I

INTRODUCTION AND SUMMARY

1. The Problem Addressed. ESDP, Evolutionary System for Data Processing, is an information system to be used to support the design, implementation and eventual modification of large computer systems. As presently conceived, ESDP would be used actively to acquire documentation of programs, data files and tests at three stages: design, after completion of the programs, and after modification. Information, once acquired, is available for retrieval, for dissemination, for incorporation in reports, for use in instruction and for use in improving subsequent information acquisition processes. The primary users of the system will be programmers, whose documentation tasks will be lightened; their supervisors, who will have more up to date information about program status than is normal; systems analysts who will be able to see the result of design change or see programming problems that force a design change; and management, who will have more current and more accurate information on progress, manpower utilization, computer utilization, etc. An instructional subsystem will help alleviate the inevitable problem of personnel turnover by providing instructional material compiled from the basic documentation at little cost in manpower.

Large programming projects are characterized by a truly staggering problem in human communication. Management decisions tend not to flow down to working-level programmers in terms most meaningful to them: What do I have to do to my program? What help will I get? How much machine time will be available next month? Almost trivial decisions made by a programmer, including implicit decisions not to do something (e.g., not to error-check an input item), can have disastrous impact on other programmers. Design errors do occur but the thought given to their resolution is not always commensurate with the magnitude of the problem. Trends toward exceeding core allocation or program running time might not be recognized until so late that massive reprogramming is needed.

Of all problems, the one ESDP is primarily aimed at solving is that of change in a system. The problem is this: given a need for a change in a program, how do we find what specific changes to make, what else in the system will be affected, how to retest the program, and how to modify the documentation.

We must emphasize that work on this project has been concentrated on support of large programming systems, where the formal documentation requirements are stringent, and the internal communication problems overpowering. While many of the techniques to be discussed would be valid if applied to smaller systems, a system of the size and scope described here is intended for the larger problem. A short discussion is given in

Section I.4 of a prototype which would not perform all functions but would be more economical for smaller systems and for further testing of overall concepts.

The ESDP system described in this report would perform the following functions:

a. Active acquisition of documentation. The system would not passively await submission of information. It would actively elicit it.

b. Storage and retrieval of documentation. All information in the system would be available for retrieval by qualified users.

c. Generation and administration of instructional material. ESDP would contain an instructional subsystem through which existing documentation could be converted into instructional form and then computer assisted instruction courses administered.

d. Production of hard copy documents. ESDP, in addition to providing a retrieval system, would produce the written documentation usually required of a programming project. No attempt should be made to replace this form of documentation. Rather, the traditional form of documentation would be augmented by the retrieval and instruction systems.

Extensions of the system can provide for:

e. Computer assisted test planning and documentation.

f. Hypothesis testing, in which system users can analyze the consequences of proposed changes to the system.

g. Computer assisted programming in which many of the basic concepts presented here are used to provide active assistance in writing object system programs (the object system is the program system being documented).

To accomplish these objectives, in addition to the further development work and programming, a using organization would have to provide a time-shared or multi-programmed computer, probably dedicated to the ESDP documentation function. If not entirely dedicated, then it would probably share time with functions other than debugging of the object program system. The preference against sharing time with the object system is based on the assumption that shortages of machine time would always be resolved in favor of the object system, even to the point of total exclusion of documentation. Also required would be an aggressive project leadership willing and anxious to try a radically new approach to an old problem, willing to encourage the use of the system and to show patience with its weaknesses.

2. The Long Term Solution. The solution to the problems of assisting programmers in the production of their programs, and in providing the needed communication and documentation, is to provide an information system with these features:

- It should be a depository for all recordable information about the object system (the program being produced).

- It should be an active system. It should take the initiative in seeking out information from programmers, analysts and managers and in disseminating that information to all concerned.

- It should be comprehensive. The system should encompass all the documentation about the object system, although not necessarily in machine-readable form.

- It should be accessible. Information can be gotten from the system by:

  information retrieval queries

  automatic report compilation

  automatic dissemination

  instructional programs.

  Users may readily shift modes--to ask for information while entering documentation, for example--so that the system is always ready to answer whatever question a user has.

- It should be self-adaptive. The information acquisition process is a function of the data base content and structure. Since the process of acquisition changes the data base, it follows that the acquisition process changes as a function of previously acquired information.

In more detail, a complete ESDP would have the following features:

a. Acquisition

Information would be acquired in two ways: by programmers or others responding to computer generated interrogations through remote consoles [1], and by programmers submitting completed programs to a program analysis [2]. Information not explicitly stored could be generated. For example, flow charts could be constructed from information acquired by either method. Changes to documentation or to programs stored within the system could be submitted through interrogation.

b. Storage

All machine readable documentation and all object programs should be stored as part of the ESDP data base. Not all this information need be continuously on-line, but all should be available, say from tape or disk cabinets, within a few minutes of the time it is requested.

c. Dissemination

A selective dissemination system should immediately transmit newly acquired information to interested readers. User interest profiles could be entered by individuals, management (on behalf of others) or even generated by ESDP. As an example of the use of automatically-generated interest profiles, assume that program A is modified to make use of an output of program B, while previously there had been no connection between the two. The programmers involved should be automatically added to distribution lists for each other's documentation.

d. Information Retrieval

Under countless sets of circumstances, system users will want to recover information immediately from the ESDP data base. An information retrieval system should be included in ESDP which will serve as an on-line system responsive to user queries and will also serve as a subsystem of other ESDP programs.

e. Generate Documentation

More or less conventional, hard copy documentation should be a major product of ESDP. The availability of a dissemination system and an on-line retrieval system will certainly have some effect on the form, content, and frequency of issue of printed reports. However, it seems impractical to assume that electronic displays will completely replace printed matter in, say, the next five to ten years.

f. Provide Instruction

The documentation contained in the ESDP data base should be convertible into instructional materials. A qualified instruction or training specialist must supervise the process, but his work would be made easier by the ESDP system itself. Both computer assisted instruction (CAI) courses and printed programmed instruction (PI) texts could be generated by a similar process. Instruction courses so generated would require far less instruction time to prepare and could be much more easily updated than conventionally-written CAI or PI courses.

g. Test Planning

ESDP should actively assist in the design, documentation, and preparation of program tests, at all levels from small program modules to large assembly tests. The information available to ESDP about a program contains much that is useful to the test designer. ESDP could assist him in planning what program segments or modules to test in any one run, what data values are to be tried, and what programs or segments are independent of what others, which information can be used to reduce the number of test runs. Finally, the system should enable the test planner to assemble a set of test data. The study of test planning was not a task under the contract. However, sufficient interest was generated to create an IBM-sponsored study in the area to be undertaken in 1968.

h. Hypothesis Testing

Related to the concept of test planning, hypothesis testing is the investigation or verification of assumptions about how the system performs. The technique is intended to be able to provide answers to questions that are phrased in external, or performance-oriented terms, rather than in terms of specific program threads or control decisions. It would be more for system users to determine how the programs would perform in a hypothetical situation than for programmers to test their programs. This topic, also, was not a part of the study just completed but is believed to be a valid extension of the study.

3. Areas of Concentration of the Study. The objective of the recently concluded study was to produce specifications for an operational ESDP where the term "specification" was interpreted as a broad, conceptual-level description. Because of the advanced nature of the project not all areas were given equal attention nor explored to the same depth. Indeed, test planning and hypothesis testing were conceived as applications during the study but not pursued as part of the study.

Major emphasis was placed on those areas which we felt were critical for proving feasibility of ESDP. These were:

Acquisition of design information

Acquisition of program documentation

On-line information retrieval

Document generation

Instruction

The acquisition of design information is to be carried out by interrogation of programmers, systems analysts or managers. It elicits their design plans for programs and data base elements. Design interrogations do not differ in concept from interrogations based upon the existence of an object program but they would be based only on information collected during previous design interrogations.

The acquisition of program documentation is, by definition, based upon the existence of an object program. It is to be done in two ways: first, by direct analysis of the program to determine its internal structure, its interfaces with other programs, and its data usage; second, by interrogation of the programmer based upon data extracted from his program. We repeat--this interrogation is not inherently different from a design interrogation but is based on knowledge of the program, hence can be much more concise and informed.

On-line information retrieval, while not a new concept, was an object of attention on this project because of the need to integrate retrieval with the interrogation and document generation functions.

Documents produced through ESDP would be hierarchically organized and primarily textual in content. They would not simply be tables, such as the cross-reference listing produced with program compilations. We have not investigated flow charting programs because of the existence of several quite versatile and effective programs with which ESDP could be made to interface [3,4]. ESDP printed documents are intended to supplant all other forms of printed documentation.

The concurrent and semi-automated generation and modification of instructional programs with the design and development of the object system is a significant new approach. It has the potential to make a major contribution toward reducing communications failures and making more efficient use of people on large projects.

4. Steps Toward an Operational Prototype. During the course of this project, both under IBM sponsorship and U.S. Air Force sponsorship, a considerable amount of experimental programming was done. The programs, most of which were done in PL/I, have never been assembled into a single, integrated system. To do so would require some additional programming and some modification to existing programs. However, these tasks could be performed

and the result would be a prototype system performing most of the functions delineated above although restricted in number of terminals, size of data base, and sophistication of the interrogation programs.

a. Programs Completed

The individual programs available now are:

(1) Program Analysis (PA). This program operates on PL/I, OS Linkage Editor, and OS/360 JCL programs only. It breaks PL/I code down to the level of a segment (a straight-line sequence of statements) and recognizes data occurrences in three categories: SET, USED, and CONTROL. While an extension of the data occurrence categories would be desirable the program is useful as it stands. The data structure compiled by PA is not now compatible with the other programs but it would be a minor task to make it so.

(2) Design Interrogation. A program exists to interrogate on program design and another on data set (file) design. These two need only minor revision to be useful as production programs. However, they were designed separately and do not interface, as they eventually should. That is, portions of the data programs should be callable from the program documentation. Both programs make use of the information retrieval system described next.

(3) Information Retrieval. Actually a package of subroutines these programs can:

- Store and retrieve data items addressed by a hierarchical number
- Extract key words from a text response and compile a key-word index of a document
- Search the key-word index, using Boolean logic, to retrieve references.

These programs represent an extension and generalization of the retrieval package associated with the original program analysis program.

(4) Report Generation. A "standard" report may be defined, by providing a sequence of hierarchical information element numbers, and a copy of this report may be generated at any time by responding "//REPORT" to any interrogation question. If a differently-constructed report is desired, the requestor may specify which information elements he wants, and in what sequence.

(5) Instruction Generation. An instruction generation program is in a sufficiently advanced state of debugging to be included in the basic package. When completed, it could be used, as well, for generation of any additional interrogation programs needed in a prototype system.

b. Additional Requirements

These five programs or groups of programs would have to be integrated and some operating rules (such as frequency of updating) devised. In addition, two additional tasks would have to be undertaken and a third is optional.

(1) Program Interfaces. In the larger ESDP, we have assumed that all documentation is available, or can readily be made available, on-line. This would be more difficult with a prototype which would be presumed to be sharing a computer with other programs. A minimum amount of information about each program in the system should be made available in a partitioned data base. This minimum documentation would contain primarily interface information so that if program A calls B, sufficient information is available on-line to provide the documentor of A the information he needs about B. Conversely, the interface data could be used to verify that the call to B was executed correctly.

To accomplish this, the ESDP data base, upon which the other programs are predicated, must be revised, and the interrogation and program analysis programs also adjusted.

(2) Multi-terminal Operations. Some additional programming is required to accommodate many terminals concurrently where all are not involved in the same activity. More than one terminal communicating with one program, even though executing different program steps, is not a significant problem. The use of PL/I which produces reentrant code plus the Queued Telecommunications Access Method (QTAM) [5] of OS/360 which buffers incoming and outgoing messages makes this possible. The problem arises when the terminals must communicate with different programs that cannot both reside in high-speed memory at once. This necessitates roll-out/roll-in for storage allocation, a feature that is not now available but will be standard in future releases of OS/360. Time slicing does not appear to be a signifcant problem at this time since ESDP activities are not likely to seize the CPU for long periods of time without being interrupted for input or output operations.

(3) Object Programming Language. In concept, ESDP is applicable to programs written in any language although some of the techniques, particularly program analysis, are specially tailored to one language. So far, the language has always been PL/I. To use ESDP with other languages, a new program analyzer would be needed and possibly some revision to the interrogation programs.

(4) Computer Requirements. The specific configuration of hardware on which ESDP is to be run depends upon such factors as the size of the object system, the documentation requirements, the philosophy of the system management, etc., and, therefore, was not considered a valid product of this study. However, since we have defined ESDP as a support system for large (25 or more programmers) programming projects, we can make some rough estimates of hardware requirements based solely on our experience and knowledge of systems of that size.

We feel that a medium to large sized CPU (e.g., /360 Mod 40-50) should be made available for ESDP. It is not necessarily true that only ESDP can be run on this CPU, but we feel that the same computer system should not be shared between the object system and ESDP.

The system should be capable of supporting a minimum of ten remote terminals, core requirements being closely related to the number of terminals active concurrently.

Bulk storage for ESDP files may be the most critical ESDP hardware requirement. Some minimum amount of file data must be on-line at all times with a larger amount off-line on disk packs, tapes, etc. A very rough guess at on-line bulk storage requirements is 30M bytes random access and 60M bytes serial access.

Until such parameters as programming language, number of terminals, and computer configurations can be established, no precise estimate of prototype production cost can be given. However, an effort of the same order of magnitude as the current contract (3.5 man years) is reasonable.

II

ACQUISITION OF INFORMATION

1. Program Analysis. Program analysis is the process by which ESDP acquires structured information on a completed program. By completed here we mean compilable, for the program analyzer does not have the elaborate error-checking mechanisms of a compiler. The object program used need not be complete in the sense of having all code written.

There are four categories of structure information. The first is concerned with the hierarchic organization of a program. In Figure 1, programs B and C are contained in, hence are hierarchically subordinate to, program A. B and C are at the same level, neither containing the other. No such condition as is illustrated by D can exist. A program is wholly contained in or wholly independent of another.

The second structural feature of interest is the branching, or control, network. This is described by listing the possible predecessors and successors of each program component. Predecessors and successors are defined in terms of execution. If program B branches to C, C is a successor of B, and B is a predecessor of C.

Category three is data structure. This is one of several instances where use of a high order language for the object programs is of great benefit to FSDP, for these languages require explicit specification of data sets. Data structure information, then, is readily available to ESDP through program analysis.

Finally, by analogy with the program control structure there is a network of data occurrences. We record in this network the mode of each occurrence of a data item name in a program element. In early work only three types of occurrence were recognized but for more advanced work, particularly in program planning, a far more intricate analysis is necessary.

In the remainder of this section, the four structural categories are discussed in greater detail.

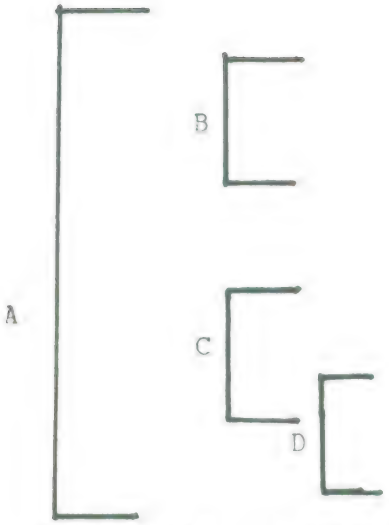

Figure 1. A Program Organization

a. Program Hierarchy

A program segment is defined as a sequence of straight line code, a set of statements or commands for which entry is only possible at the first statement and branching is only possible from the last statement. This is the smallest unit of programming considered in work to date but we could go to the individual statement without changing the substance of this description. A unit of programming is a program segment or any set of program segments. Thus, a unit of programming (UOP) can be as small as a single IF statement or as large as Operating System/360. In general, we are concerned with acquiring and publishing the same kind of documentation about a UOP at any level but , in practice, we would vary the information with the nature of the UOP.

The program hierarchy is represented by a set of subordination relationships in the documentation record for each UOP. For each UOP there will be a list of subordinate and superordinate UOP's (downward and upward pointers).

b. Program Control Network

A successor of a UOP is any UOP, at any level, that may be executed immediately after the first one. The number of possible successors may be quite large, as might happen if a UOP in PL/I ends with GO TO LABEL(I) when LABEL is the name of a label array. The number of possible successors is the number of elements of array LABEL. On the other hand, GO TO START has only one successor.

The knowledge of what data variables, if any, control the selection of the successor is important information in documentation, debugging and test planning.

The program control structure is stored, as is the hierarchy, by including two lists in each UOP documentation record. One list contains successors and one predecessors.

c. Data Structure

Data structure is generally more complex than program structure. The language of data description tends to be dependent upon the programming language used. In general, what is needed is the name of the data element, its type (whether record, array, field, etc.), and its subordinate and superordinate elements. In the case of arrays, the number of elements is important and in the case of individual items (or fields or variables) such information as:

Mode or type (fixed decimal, character floating binary, etc.)

Length

Initial value

A data set, or file, is not fully described by its hierarchical structure. The nature of the relationships between elements, say two variables in a structure, must be made known, but this cannot be determined through program analysis.

d. Data Occurrence Network

This information structure tells where data is used, where it is changed, where it affects a branching decision. For any UOP its data occurrence table represents a "black box" description of that UOP--a statement of the input, the output, and the variables governing choice of the next, or successor, UOP. Table I shows a far more detailed system for describing data occurrences than has been used in experimental work. It should be understood that a given data element can occur in more than one category within any UOP.

The complete program structure is describable in terms of the record outlined in Table II.

Table I. Classification of Data Usage by a Program

1. Context of Appearance

   1.1 Assignment Statement
       1.1.1 Computed Value
       1.1.2 Argument

   1.2 Control Statement
       1.2.1 Variable I/O Command
       1.2.2 Branching or Transfer Command
             1.2.2.1 Argument or condition statement (IF, ON ...)
             1.2.2.2 Iterative Control Variable (DO)
                     1.2.2.2.1 Initial index value
                     1.2.2.2.2 Increment
                     1.2.2.2.3 Maximum value or limit
             1.2.2.3 Variable address

   1.3 Subroutine/Function/Macro Calling Sequence
       1.3.1 Transmitted to SR/Function/Macro
       1.3.2 Received from SR/Function/Macro

   1.4 Data Declaration Statement (or other non-executable statement)

   1.5 Input/Output
       1.5.1 Input
             1.5.1.1 Input Control Variable
             1.5.1.2 Data Element read in
       1.5.2 Output
             1.5.2.1 Output Control Variable
             1.5.2.2 Data Element written out or transmitted

2. Change Status

   2.1 Value Changed by Containing Statement
       2.1.1 Value Directly Assigned by Assignment Statement
       2.1.2 Value Directly Changed by DO Statement
       2.1.3 Value Directly Changed by Variable I/O Statement

   2.2 Value not Changed by Containing Statement

3. Structural Role

   3.1 Data Element is a Structure or Array

   3.2 Index or Subscript
       3.2.1 Value of an Index
       3.2.2 Element of an Index Term

   3.3 Scalar Item

Table II. Program Report Outline

1. Identification

1.1 Author

1.2 Name of Program or System

1.3 Document Modification Data

1.3.1 Modification Number

1.3.2 Author of Modification

1.3.3 Date of Modification

1.3.4 Date of Initial Entry

1.4 Level of Documentation

1.4.1 Designation

1.4.2 Program Type

2. Introduction

2.1 Program Summary

2.1.1 Description of Logic

2.1.2 Program Application

2.1.3 Program Organization

2.2 Entry

2.3 Exit

2.4 Data

2.4.1 General Summary

2.4.2 Inputs

2.4.2.1 Summary of Inputs

Table II. Program Report Outline (Continued)

2.4.2.2 Input Files

2.4.2.2.n.2 Definition

2.4.3 Outputs

2.4.3.1 Summary of Outputs

2.4.3.2 Output Files

2.4.3.2.n Output File Number

2.4.3.2.n.1 Name

2.4.3.2.n.2 Definition

2.4.4 Internal Data

2.4.4.1 Summary of Internal Data

2.4.4.2 Internal Files and Data Sets

2.4.4.2.n Internal File Number

2.4.4.2.n.1 Name

2.4.4.2.n.2 Definition

3. Program or System Functions

3.1 Description of Logic

3.2 External References

3.2.n Reference Number

Table II. Program Report Outline (Continued)

4. Program or System Composition

4.1 General Description of Structure

4.2 Subordinate UOP

4.2.n Subordinate Number

4.2.n.1 Name

4.2.n.2 Function

4.2.n.3 Inputs to Subordinate

4.2.n.3.n Input Number

4.2.n.4 Outputs from Subordinate

4.2.n.4.n Output Number

4.2.n.5 Internal File of Subordinate

4.2.n.5.n Internal File Number

4.2.n.6 Entry-Exit Conditions

4.2.n.6.1 Summary of Entry Conditions

4.2.n.6.2 Summary of Exit Conditions

4.2.n.6.3 Type of Decision Governing Exit

4.2.n.6.3.1 Successor (if unconditional)

4.2.n.6.4 Exit Control Variables (if conditional)

4.2.n.6.4.n Control Variable Number

4.2.n.6.5 Successors to Subordinate (if conditional)

4.2.n.6.5.n Successor Number

4.2.n.6.5.n.1 Name

4.2.n.6.5.n.2 Control Conditions Causing Path

4.2.n.6.5.n.3 Purpose of Taking Path

Table II. Program Report Outline (Continued)

5. Control

5.1 Entry

5.1.1 General Description of Entry

5.1.2 Entry Points (if program level)

5.1.2.n Entry Point Number

5.1.2.n.1 Identification

5.1.2.n.2 Conditions for Using Entry

5.1.2.n.3 Subordinate Containing Entry

5.1.2.n.4 Predecessors

5.1.2.n.4.n Predecessor Number

5.2 Exit

5.2.1 Exit Conditions

5.2.1.1 General Description of Exit Conditions

5.2.1.2 Type of Decision Governing Exit

5.2.1.2.1 Successor Program or System (if conditional)

5.2.1.3 Description of Decision Functions

5.2.1.4 Variables Controlling Decision (if conditional)

5.2.1.4.n Variable Number

5.2.1.4.n.1 Name

5.2.1.4.n.2 Definition

5.2.2 Successors (if conditional)

5.2.2.n.1 Name

5.2.2.n.2 Control Conditions Causing Path

5.2.2.n.3 Purpose of Taking Path

Table II. Program Report Outline (Concluded)

6. Data

6.1 General Summary

6.2 Inputs

6.2.1 Summary of Inputs

6.2.2 Input File

6.2.2.n Input File Number

6.2.2.n.1 Name

6.2.2.n.2 Source

6.2.2.n.3 Description of Content

6.2.2.n.4 Structure

6.3 Outputs

6.3.1 Summary of Outputs

6.3.2 Output Files

6.3.2.n Output File Number

6.3.2.n.1 Name

6.3.2.n.2 Destination

6.3.2.n.3 Description of Content

6.3.2.n.4 Structure

6.4 Internal Data

6.4.1 Summary of Internal Data

6.4.2 Internal Files and Data Sets

6.4.2.n Internal File Number

6.4.2.n.1 Name

6.4.2.n.2 Description of Content

6.4.2.n.3 Structure

2. Interrogation

Computer Assisted Interrogation (CAINT) is a computer program system for man-machine communication.

> Its principal function is to enable a computer to elicit information from a man by interrogating him--asking him a program of questions where the program follows a logical course depending both on information available before the interrogation started and on that gained during the interrogation. The information acquired is intended to be put to immediate use, in updating a data base, generating reports, or driving other interrogations.
>
> During the course of an interrogation, the interrogee will be given information as well as asked questions, and he may ask his own questions as well as provide answers. Thus, a CAINT interrogation is truly conversational, with information and questions flowing in both directions. The conversation, particularly in the machine-to-man direction, is somewhat stereotyped, the machine's versatility being limited by a repertoire of generalized, fragmented statements which are particularized and assembled for use as needed. The conversational range of the computer, then, depends upon a system user's versatility in designing these statements -- a process somewhat akin to computer programming. [1]

In planning an interrogation program the designer must constantly keep three objectives in mind: (1) that of acquiring the set of information that best accomplishes the documentation of a program, (2) that of assuring that the interrogation program is thoroughly debugged, and (3) that the programmer or other user who is responding to an interrogation understands the manner in which elicited information is to be used. It should be clear that the successful application of ESDP to any project depends heavily upon the degree to which these objectives are met. However, they are essentially the same requirements imposed on the development of any computer program.

The need for skill and accuracy on the part of an interrogation programmer is no different from the same need in conventional programming. Interrogation programming, as we shall bring out, can be mechanically easier than conventional programming, but properly identifying objectives and ensuring proper user interface may be more difficult. Because this is the primary channel for the entry of much information into ESDP, it is highly important that users have skill in responding to

interrogations and in interpreting questions in terms of their own programs.

> Guided by the structure of the object program, and design information previously contributed by the programmer and others, CAINT interrogates a programmer about his own program and elicits a detailed, up-to-date, program description. It can also be used for other highly structured reporting activities and for advanced educational programs. [1]

CAINT, which was developed as a subsystem of ESDP enables programmers to document programs while still at the design level and to add elaboration and explanation to documentation acquired by program analysis. As we have pointed out earlier, the process of interrogation is not logically different for those two modes of operation. All that changes is the wording of questions, and possibly the choice of which questions to ask.

An interrogation is based upon a data structure. The object of interrogation is to create, complete or modify this structure. To this end a set of questions and question fragments are prepared and a program is written which decides which questions or combination of fragments to use, asks the assembled question, analyzes the response and, if valid, stores the response in the data structure. In deciding what question to use and in analyzing a response, the program may make use of any programmable function of the data structure or of any other data structure available to the program. Such a program is called an interrogation program and it is written in the CAINT Executive Language.

More details of interrogation will be brought out below. At this point we wish to emphasize the following principles:

a. To carry out an interrogation, an interrogation program must be written. This program uses any function of the data base to make decisions, and results in acquiring new information for storage in the data base.

b. Interrogation programs are intended to be readily modified. A basic assumption is that each using organization will have different needs or preferences from each other organization, and CAINT is designed to accommodate these differences.

c. Interrogation was originally conceived to operate upon the results of program interrogation--to ask the programmer for explanation of the structure of his program. Further investigation suggested the value of interrogation as a design tool, for interrogating programmers about programs before their completion, and as an instructional tool for presenting existing

information to the programmers.

3. Design Interrogation Program documentation is most valuable when it is available to assist decision-makers and designers. The point of greatest need for this assistance comes early in a project, when there are no completed programs. It is during the first few months when program interfaces are being specified, files being designed, and schedules being laid out that valid current documentation would be of most value and is least likely to be available.

Even at the earliest stages of system development, designers will have ideas in mind about general program structure, program interface and file structure. The fact that these initial ideas will change in no way invalidates the need for recording them and disseminating them as they are created.

Design interrogation has the same objective as program analysis and interrogation--the acquisition of information about program and data structure and explanatory text. Even so primitive a program model as INPUT-COMPUTE-OUTPUT, if coupled with a list of files used and produced, and other programs communicated with, can be very useful, particularly if such a document is created for each system program or module. To this we can add information on programmers assigned and major milestones, and then the skeleton of a system plan begins to form.

The original concept of ESDP assumed the full use of design interrogation, even in the absence of a program specification. The intent was that a specification would evolve through repetitive interrogation, and that when the program was eventually written, it would match well with the design data in structure. The successful use of this concept relies heavily upon the cooperation and probably enthusiasm of its users, for they must reply to necessarily vague questions, in order to introduce the program structure to the system, and must anticipate the use to be made of their responses in order to improve future interrogations. It is possible that design interrogation will be more effective, at least during the exploratory phases of developing ESDP, when applied to data rather than to programs. Data files are definitely of interest to more than the originating programmer, their design tends to change often, and changes can have major impact on other users. Hence, all the communication problems of the system as a whole exist for data as a subsystem, but the manner of describing data structures is perhaps more consistent than for programs, and the information needed about a file under design is more precise and generally better agreed upon than for programs. Hence, an option open to systems managers is to use design interrogation for both programs and data, for data only, or neither.

Design interrogations if used must be repeated at frequent intervals. Once the basic framework of the system is laid out conflicts and omissions become evident which lead to an

almost endless series of changes. Gradually, however, the structure should begin to stabilize and the resultant documentation should, ideally, be identical to that to be acquired through program analysis. That there will be differences between plan and execution is certain and for this a reconciliation process (Section II.6) has been devised.

Automatic dissemination of design documentation can be a powerful management aid. All programmers, designers and managers can see, readily, the current status of the entire system. Even more significantly, they can immediately be apprised of any design changes related to their own areas of work.

The same interrogation technique used for acquiring technical data could be used for acquiring program status and other management information. Thus, changes in design and other factors could be immediately translated into changes in plans.

Figure 2 is an excerpt from a design interrogation. The interrogation was carried out on an IBM 2260 CRT display terminal. The text of the first few questions has been retyped for legible reproduction here.

Figure 3 is a copy of the report generated as a result of the interrogation illustrated in Figure 2. This is an abbreviated report which results from using a short interrogation program that was primarily designed for experimenting with interrogation techniques.

```
THIS IS A PROGRAM DESIGN INTERROGATION  1/1/68

WHAT IS YOUR NAME?
     JOHN DOE

WHAT IS THE NAME OF THE UOP YOU WISH TO DISCUSS?
     ALTAIRFLD

DESCRIBE THE PURPOSE OF THIS UOP
     THIS IS AN ILLUSTRATIVE PROGRAM CONCEPT TO DEMONSTRATE ESDP
     TECHNIQUES.  IT IS BASED ON A FUNCTION THAT MIGHT BE PERFORMED
     BY AN ACTUAL AIR TRAFFIC CONTROL SYSTEM.  IT USES PARAMETERS
     FOR AN INDIVIDUAL AIRCRAFT, TOGETHER WITH A STORED FILE OF
     AIRCRAFT CHARACTERISTICS, TO COMPUTE THE DISTANCE THE PLANE
     CAN FLY WITH ITS REMAINING FUEL.  THEN IT SELECTS THE FIRST
     ALTERNATE AIRFIELD THAT CAN BE REACHED.

SELECT THE NUMBER THAT BEST DESCRIBES THE LEVEL OF ALTAIRFLD:
1. SYSTEM, 2. JOB, 3. LOAD MODULE, 4. OBJECT MODULE,
5. CALL MODULE, 6. GROUP, 7. SEGMENT
     5

SELECT THE LETTER OF THE TOPIC YOU WOULD LIKE TO DISCUSS
A. PURPOSE OF ALTAIRFLD, B. PROGRAM STRUCTURE, C. PROGRAM
CONTROL, D. DATA, E. ERRORS, F. TESTING
     B

DO YOU KNOW THE NAME OF A SUPERORDINATE UOP CONTAINING ALTAIRFLD
IF SO, GIVE THE NAME, OTHERWISE ANSWER NO
     TRAFFIC
```

Figure 2. SAMPLE INTERROGATION

```
WOULD YOU LIKE TO A. CONTINUE WITH ALTAIRFLD OR B. DISCUSS TRAFFIC
   A

ARE THERE ANY SUBORDINATE UOP CONTAINED IN ALTAIRFLD?
   YES

LIST ALL THE UOP THAT ARE IMMEDIATELY CONTAINED IN ALTAIRFLD +LIST+
   INITIAL
   DISTCALC
   SELECT
   OUTPUT
   //END

WOULD YOU LIKE TO TALK ABOUT:
   1. INITIAL
   2. DISTCALC
   3. SELECT
   4. OUTPUT
   5. OR CONTINUE WITH ALTAIRFLD
   5

NOW SUMMARIZE THE STATIC ORGANIZATION OF ALTAIRFLD, SHOWING THE
RELATIONSHIP OF IT TO TRAFFIC PLUS THE RELATIONSHIP OF:
   INITIAL
   DISTCALC
   SELECT
   OUTPUT
TO ALTAIRFLD
   ALTAIRFLD IS INVOKED BY A CALL STATEMENT IN TRAFFIC.  WITHIN
   ALTAIRFLD, INITIAL, DISTCALC, SELECT, AND OUTPUT ARE INVOKED
   SEQUENTIALLY IN THAT ORDER.
```

Figure 2. SAMPLE INTERROGATION (Concluded)

TITLE PAGE INFORMATION

1 UOP NAME
ALTAIRFLD

2.1 LEVEL
ALTAIRFLD IS A CALL MODULE UOP.

2.2 AUTHOR
JOHN DOE

2.1 DATE
680118

EXPLANATION OF FUNCTION

3 PURPOSE
THIS IS AN ILLUSTRATIVE PROGRAM CONCEPT TO DEMONSTRATE ESDP TECHNIQUES. IT IS BASED ON A FUNCTION THAT MIGHT BE PERFORMED BY AN ACTUAL AIR TRAFFIC CONTROL SYSTEM. IT USES PARAMETERS FOR AN INDIVIDUAL AIRCRAFT, TOGETHER WITH A STORED FILE OF AIRCRAFT CHARACTERISTICS, TO COMPUTE THE DISTANCE THE PLANE CAN FLY WITH ITS REMAINING FUEL. THEN IT SELECTS THE FIRST ALTERNATE AIRFIELD THAT CAN BE REACHED.

BASIC FORM OF THIS UOP

4.1 SUMMARY
ALTAIRFLD IS INVOKED BY A CALL STATEMENT IN TRAFFIC, WITHIN ALTAIRFLD, INITIAL, DISTCALC, SELECT, AND OUTPUT ARE INVOKED SEQUENTIALLY IN THAT ORDER.

4.2 SUPERORDINATE UOP
TRAFFIC

4.4 SUBORDINATE UOPS

	SUB-NAME
INITIAL	1
DISTCALC	2
SELECT	3
OUTPUT	4

Figure 3. ESDP-CAINT PROGRAM DOCUMENTATION FOR ALTAIRFLD

LIST OF MODIFICATIONS TO THIS UOP

2.5 MODIFICATIONS

ALTAIRFLD--REPORT COMPLETED

ACTUAL	IEN 3.00		
AIR	IEN 3.00		
AIRCRA	IEN 3.00		
AIRFIE	IEN 3.00		
ALTAIR	IEN 4.01		
ALTERN	IEN 3.00		
BASED	IEN 3.00		
CHARAC	IEN 3.00		
COMPUT	IEN 3.00		
CONCEP	IEN 3.00		
DISTAN	IEN 3.00		
DISTCA	IEN 4.04	IEN 4.01	
ESDP	IEN 3.00		
FILE	IEN 3.00		
FLY	IEN 3.00		
FUEL	IEN 3.00		
FUNCTI	IEN 3.00		
ILLUST	IEN 3.00		
INDIVI	IEN 3.00		
INITIA	IEN 4.04	IEN 4.01	
INVOKE	IEN 4.01		
ORDER	IEN 4.01		
OUTPUT	IEN 4.04	IEN 4.01	
PARAME	IEN 3.00		
PERFOR	IEN 3.00		
PLANE	IEN 3.00		
PROGRA	IEN 3.00		
REACHE	IEN 3.00		
REMAIN	IEN 3.00		
SELECT	IEN 3.00	IEN 4.04	IEN 4.01
SEQUEN	IEN 4.01		
STORED	IEN 3.00		
SYSTEM	IEN 3.00		
TECHNI	IEN 3.00		
TOGETH	IEN 3.00		
TRAFFI	IEN 3.00	IEN 4.01	

Figure 3. ESDP-CAINT PROGRAM DOCUMENTATION FOR ALTAIRFLD (Concluded)

4. Changes in Documentation. Changing existing documentation will be the normal mode of operation of ESDP once an initial program description has been entered. A programmer will not be asked to repeat previously entered information if he just wants to enter a change. Changing operates on the principle that the programmer describes the information element or program element that is to be changed and analysis or interrogation will be performed only on the stated items.

In interrogation the programmer identifies an information element by number--this being a hierarchic tag applied to each separable item of information in the files. The author of the interrogation program will have compiled a list of questions pertaining to that element. Only these questions will be asked again and within these prescribed limits, the reinterrogation may follow a different sequence of questions than the original. The programmer is not restricted to making changes at the bottom of the hierarchy. He may specify a high-order element which contains many subordinates, and if he does he will be reinterrogated about all of them.

Program analysis will also operate on an incremental basis. In this case, the programmer identifies the lowest order UOP not affected by the change and it, and all its members, but no other program components, will be reanalyzed. Here, a problem is created. The structure of the program may be so changed that ESDP is unable to match old UOP's with new, hence cannot treat newly computed data as changes to an existing structure. Instead, a parallel structure must be created and the two merged later by a man-machine process.

5. Incremental Documentation. The term incremental documentation refers to ESDP's ability to accept changes to existing documentation without completely regenerating it. It is a feature of ESDP that facilitates production of current documentation of an object program system as it evolves, reflecting the current status of programming through the stages of design, debugging, and modification, with a minimum of effort by the user.

a. Design Phase

The information in the ESDP documentation files during the design phase of the object system is entirely derived through interrogation. Since the files contain no data derived through program analysis, the incremental documentation for this phase is wholly concerned with incremental interrogation.

The purpose of incremental interrogation is to enable documentors to make changes in documentation files or documents with a minimum of effort and, at the same time to ensure that when a change is made, all consequences of that change are also effected. For example, if we were to change the type of a data item from, say, variable to array, it would be necessary to make some other implied changes as well, relating to such attributes

as length of the array which would not previously have been present in the file.

The method, in general, is to write the interrogation program so that each interrogation element (programming statements concerned with a question on the elicitation of a single information element) can be operated as a subroutine or as a normal part of a normal program. This is done by inserting a test at the end of the element to see if the program is being operated in the subroutine or sequential mode. If the former, transfer is made to the calling program; if the latter, control passes to the next sequential element.

In experimental interrogation we have used a subroutine linkage, CALL NEXT, as the test at the end of each element. The NEXT subroutine determines if the responder is involved in an update activity. If so, the element has been used as a subroutine and control is passed by NEXT back to the point from which the element was called. Otherwise, control will be passed to the next sequential element via a RETURN statement.

The control logic for incremental interrogation essentially revolves around a table of information element numbers (IEN's) or documentation items, and the interrogation elements that must be used in order to update the IEN. The user inputs the IEN, the system looks that number up in the table, retrieves the list of interrogation elements that must be executed, and carries out the interrogation. The newly elicited information is later merged into the existing documentation files.

This description has assumed the documentor knows what IEN's he wants to modify. Such is not always the case. If he can define the area of interest only in terms of subject matter, he may use the information retrieval system to find those IEN's that are related to his known subject. He could, then, start an interrogation with a request to update all documentation pertaining to input operations from tape units, retrieve the appropriate IEN's, and then proceed normally.

To carry out a document modification, the system user must identify the subject concerned, whether a Unit of Programming, Unit of Data, Graphic, etc., and the name or number of that entity. He must then identify the document from which he is making up this modification. This will have to be some identifiable report, flow chart, retrieval output, or even message on a CRT, but the system must know what is to be modified. Next he enters the change command, whether add, delete, or change, and then enters new information in response to the ensuing interrogation questions.

There is some value in holding changes in a separate file, at least temporarily, until a change session is completed and the author is satisfied he has made all the changes correctly. For system reliability purposes, we should minimize

the possibility of a system failure occurring during the posting of a change to a file. For this reason, it is better to make the complete change, hold it in working storage, then apply it all at once to the main file.

There are two tables involved in the determination of what interrogation elements to use for an incremental interrogation. The Sequence Table (illustrated in Figure 4) provides a list of interrogation elements for each information element number (IEN). This list, in work so far, has been in the form of a set of start and end points for contiguous interrogation elements, i.e., for a given IEN, we are directed to start at interrogation element a and continue sequential operation of the interrogation program until element b is reached, at which time return is made to the sequencing program. There may be any number of such start-end pairs. There may be any number of such start-end pairs.

The next list is the one used to convert from report section numbers, for any ESDP report, to information element numbers, for use in the Sequencing Table. This requires the maintenance of a dynamic file that records what information elements were used and under what report section number, for each report produced. Since many reports will be working papers only, the records of these reports will have to be purged periodically, and when this happens any subsequent changes would have to be made by using a different report as the basis for regenerating the report.

Report section of information element numbers may have indexable components. For example, if Section 1.2 is concerned with system inputs, the first system input might be described in Section 1.2.1, the second in 1.2.2, etc. Then, if the description of an input starts with its name, the name of the first input might be filed in element 1.2.1.1, the name of the second in 1.2.2.1, etc. The third component of these index numbers is the index of the inputs, and we call this component of the IEN indexable.

In the sequence table, it is neither necessary nor, for space reasons, desirable to have an entry for each possible value of an indexable IEN. Accordingly, we have used the technique of replacing an indexable number in this table with the letter n. Thus, the IEN for system inputs might be represented in this table as 1.2.n, and for all input names as 1.2.n.1. In matching an IEN entered by a documentor, the system treats the n as a universal match character, and accepts any numeric value for this component within a legal range for this number. For example, if he declares his intent to change the description of a data element identified by IEN 1.2.3, the system would respond, "This is item SALARY. Is this the one you want to change?"

IEN	Start	End	Start	End
1.1	Label 1	Label 2	Label 3	Label 4 ...
1.1.1	Label 5	Label 6	...	
1.1.2				
1.2				
1.2.n				
1.2.n.1				
1.2.n.2				
1.2.n.2.n				

.

.

.

Note the use of the symbol n whenever an element can repeat, with a different IEN each time. Note also, that there may be any number of indexable components of any IEN

Figure 4. Sequence Table

b. Debugging and Modification Phases

Once code has been written for the object system, ESDP must contend with both the data derived from interrogation and that derived from program analysis. When confronted with the need to modify documentation files to reflect a program change, ESDP will have in its files the following kinds of data.

(1) Structure

Program structure will be reflected in two dimensions, hierarchy and process flow. Each UOP may be contained in higher order, superordinate UOP and may also contain subordinate UOP. The process-flow dimension relates UOP by possible flows of the execution of the program. Each UOP may receive control from predecessor UOP and may pass control to other successor UOP.

(2) Formal Text

This text comprises the original source language object system program statements.

(3) Data References

The data names used in each UOP will be listed. In addition to the names themselves, there will be indicators depicting attributes and special codes categorizing the data usage.

(4) Narrative Text

The narrative text that has entered the system via interrogation is in the ESDP files. Each element of text is tagged with a variety of identifiers such as the question number that elicited the text, IEN number, keywords, associated UOP, or data names, etc.

(5) Reports

There may also be a number of reports on file. These will essentially be rearrangements of the narrative text into various report outlines. In the reports, each element will be identified by the report number plus IEN.

Adding and deleting source program statements can be automatically reflected in the structure files. This is done through manipulation of pointers. At the same time, however, it appears that at some point of modification complexity it will be more economical to rerun the entire program analysis. That is, if a great deal of the program structure is changed it may be desirable to flush all data with a program analysis source code and resubmit the program for program analysis.

If we process changes to source programs to update structure files, we are left with the situation where the program

analysis data has been updated automatically and the narrative text has not. At this point we need to link applicable elements of text to the program analysis structure, interrogate on new portions of structure, reinterrogate to elicit changed narrative or to delete narrative. These operations will be discussed below under II.6, Reconciliation. Our solution to this last step of linking stored textual data with updated portions of structure is through use of the reconciliation commands.

6. Reconciliation. Reconciliation is the process of combining information from more than one source that is descriptive of the same entity. The assumption is made that there are discrepancies in the information provided by the different sources. If there were no discrepancies, reconciliation would be an almost trivial problem, largely one of merging records. Discrepancies will be in many instances unexpected by the programmer. He will expect a difference when he knows he has written his program to be in some way different from his current design documentation. Such a difference simply reflects his failure to update design documentation. An unexpected difference arises when the programmer unknowingly violates his design specifications. His problem here is recognizing that differences exist and correlating those elements of each data record that do coincide.

Perhaps we should also point out what reconciliation is not intended to achieve. It is not the intent of ESDP to attempt automatically to correlate finished programs with design specifications to determine if specifications have been met. Although this would be an interesting and useful feature this type of correlation is a highly subjective process making mechanization infeasible. The purpose of reconciliation in ESDP is to avoid complete duplicate entry of information on the part of the user. If the system has acquired information about a program via design interrogations, it is reasonable to assume that at least a large part of this information will still be valid once the program is written and, therefore, should be tied in some way to the files built by program analysis. The information acquired during design interrogation may not include all of the details known by the programmer at the time of program analysis. Therefore, an interrogation following program analysis will be conducted to provide more details and to supply changes to the design information.

Reconciliation problems exist in regard to data documentation as well as program logic, since many programmers may use the same data item and each may define it and name it differently. The analyst responsible for system-wide data specification and control must reconcile data documentation acquired from a large number of sources; from design interrogation or program analysis on each program that uses or makes reference to a data element. This analyst, unlike the individual programmer, cannot assess the differences he sees from the different sources and resolve conflicts at once. To resolve a difference in interpretation of a data element by two or more programmers, he must talk to them and assess the impact of a

change on each program and on the overall system.

Data base control analysts may add a new version of data documentation, either to combine and amplify other documentation or to announce the existence and causes of conflicts.

This notion of standardized, approved data items brings up another consideration. There may be a need for two different kinds of documentation; one that reflects the current version of the program and one that reflects the approved version. During the evolution of the system of programs, there could be a requirement for documentation of a program as it exists currently regardless of whether it still has bugs in it and whether it uses approved data definitions; there is also a need for documentation of the design of the program using standardized data definitions. Therefore, it seems desirable to retain some degree of duplication throughout the evolution of the object system although, technically, reconciliation could take place early in the project life.

At first glance, it seems that the solution to the reconciliation problem would be one that is completely automatic. However, consider the case where two program structures (UOP), one derived through program analysis and the other derived through design interrogations, are identical. The UOP names are the same, the subordinate and superordinate elements are the same and the data usage indicators are the same. But it is still possible that the logic is different. For example, if the design specifies that Y = X/2 but the coding is Y = 2/X, then branching decisions based on the value of Y relative to 1 might be inverted, i.e., if Y $<$ 1 the program would execute the UOP specified for Y $>$ 1.

For another example, assume this hierarchical structure of a UOP illustrated in Figure 5. This indicates a Job containing three Load Modules, one of which contains three procedures. Further assume that the file created through design interrogation shows this structure using the indicated names and associated text descriptions, but has no more structure information. The file created through program analysis, of course, goes into a finer substructure but to this point matches the interrogation-acquired file. It seems certain that this match is not sufficient to conclude that text accumulated to describe, say, Load Module 1 will necessarily describe Load Module 1 as it has been coded.

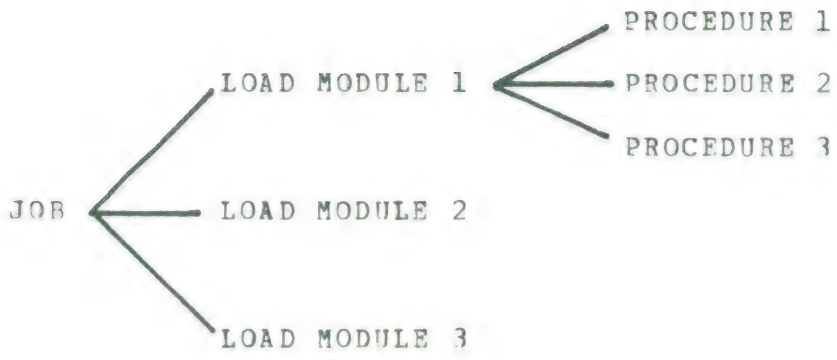

Figure 5. A Job Structure

III

DISSEMINATION AND ANALYSIS

1. Information Retrieval. ESDP produces two forms of documentation--hard copy reports and machine readable files. The later are available for search by an on-line information retrieval system. We foresee the retrieval system being used either for short-reply queries or occasional special listings. Otherwise, conventional reports will probably be used. The actual balance in use of these two programs will depend on user preference with an operational system.

Perhaps the most important use of the retrieval capability will be to assist programmers who are being interrogated or instructed. In either case, a terminal operator will be able to interrupt the programmed conversation to ask a question, perhaps to check the specification of a data item or to review an earlier entry of his own. Therefore, one of the key requirements of the ESDP retrieval system is immediate responsiveness. Another is that the IR system operate as a subroutine of any interrogation or instruction program. Occasionally, the logical niceties of the IR system design have to be subordinated to these dominating requirements.

In pure documentation applications, the logical requirements of IR are rather modest. We anticipate that most queries would call out a program element or data item by name and ask for some attribute of it or, at most, would step through a network of program elements, using the IR system as a self-instructional tool. In other uses of ESDP, though, more elaborate retrieval logic is necessary. When using the system for test planning, or for browsing to find where and how a proposed change might have to be made, we would expect more of a requirement than for single, one-item queries. It would seem that a multifile search capability is clearly indicated in this mode of operation. Multifile, in this sense, is used to mean that information from more than one file is needed to evaluate a search criterion. Most multifile queries could be accomplished through a single file search system, with enough time and patience, but lengthy search processes are disruptive of a high intensity browsing operation.

The files to be searched consist of descriptions of

Units of programming

Units of data

Units of instruction

Program text

Management information

Report text and outlines

Working storage

File directories.

The first three represent networks and hierarchies showing structure of programs and data, and interconnection among structures. Each element of information will have a unique identifying number, called an information element number, or IEN. (See Section II.5)

A table of IEN's will be stored for each standard report to be generated. This table would list the IEN's to be included and the order of inclusion. Each information element in a report would have an element number of the same form as used for IEN's but meaningful only in context of the report. For each report then, there would be a table of equivalences between report IEN's and system IEN's. A report covering inputs to all system programs might renumber IEN 1.2 for program 15 and assign a report IEN of 3.7.15.

Unplanned reports can be generated at any time by providing a list of IEN's to be retrieved.

2. Selective Dissemination. We repeat here an earlier assertion: selective and automatic dissemination of design notes and changes might be the greatest service of ESDP during the design and production stages of a programming project. Selective dissemination [6] is not a new concept and needs little elaboration here except to point out that we consider activity description as well as subject descriptions to be important in dissemination decisions. This means that a change to IEN 1.2.3, regardless of the subject matter of the change, would be disseminated to anyone interested in IEN 1.2.3.

3. Instruction. On a large programming project, turnover of people and the coming together of people from many different disciplines creates a training need which is rarely fulfilled. What is needed is a training program, tailored t o use by individual students, that can be used to indoctrinate new people in the system. This requirement is made more difficult by the frequent changes in program documentation and by the different levels of prior knowledge of the system on the part of the students.

A computer assisted instruction approach has been developed which makes maximum use of existing program documentation and is quite flexible in regard to changes in course content. The basic approach is to use CAINT as a vehicle for composing CAI courses. A course author converses with an ESDP program called the Instruction Generator and, at the completion of the conversation, an instruction course written in

the CAINT Executive Language is produced. During the conversation, the author can easily have the generator extract whatever information he wants from existing documentation.

The generator program could be written to produce either computer assisted instruction courses or printed, programmed instruction courses.

Use of this technique enables training specialists to compose and maintain up to date, self-instructional material on all aspects of the object system--whether or not documentation is complete. New personnel, or people being transferred within a project can learn new functions and bring themselves up to date at relatively little cost to the project.

The instructional system is described in greater detail in Volume 3 of this report.

4. Management Analysis. We make the following assumptions when considering the application of ESDP to programming management.

a. Better documentation and dissemination will extend and improve management control over the project.

b. Management can collect better status and projection information through interrogation than otherwise. This information would be more up to date, more precisely relevant to the actual problems of the moment, and consequently can reduce problems caused by failure to communicate.

c. Having most management data in machine readable form gives the opportunity for continual review, recomputation of budgets, schelules, resource allocations, etc.

Part of any interrogation, and this is one of the strongest reasons for having design interrogation on programs, should cover the history, the status, and the prospect for that program. One of the great weaknesses of present day management reporting systems is that people using them are free to report the same information over and over, without detection. For example, programmers are notorious for reporting that programs are ninety per cent debugged. Relatively simple programs can analyze the pattern of responses to management queries and elicit the necessary explanations. This is not to imply that a programmer cannot be honestly convinced, on each of five successive weeks, that he is ninety per cent done. However, feeding his own history of responses back to him and asking for his interpretation may lead him to recognize that his estimates are inaccurate, and, hopefully, lead him to improve accuracy. In general, the power of CAINT can and should be used to elicit from responders the nature of any problem, recommendations for its solution, and should provide reviews of previous estimates and problem descriptions. This assures that information on management problems is written down and is actually transmitted to those responsible. It also assures subordinates that their

problems and recommendations are reaching their managers.

The information to be collected relative to program management is variable, as is program-descriptive information, and is to be determined by the using organization. The following list, though, is representative:

Identification of objective or milestone (e.g., completion of a program, a test, a flow chart ...)

Time: estimate to complete task

actual time spent to date

review of previous estimates

explanation and discussion of discrepancies or inconsistencies

Manpower: estimate for entire task

manpower expended to date

estimate to complete

review of previous estimates

explanation and discussion of dis-discrepancies or inconsistencies.

Related milestones or tasks, information on other tasks dependent upon or prerequisite to this one which might be affected by a schedule change, or for which there has been a schedule or other change.

Special problems being faced, decisions needed from higher management, recommendations on these problems, constraints imposed by management or other factors such as overall cost limitations final completion deadlines, etc.

We must stress that, while ESDP provides the communication medium for the exchange of this information, and while it can efficiently collect far more detailed managerial data than is usual, it remains with the project managers to make the best use of this information. Certainly, any laxity on their part, or tendency to ignore problems reported in this manner, or tendency not to take seriously ESDP as a management tool will be immediately reflected in the work of others, possibly extending into technical documentation.

IV

EXTENSIONS OF THE BASIC SYSTEM

1. Introduction. The original concept of ESDP was for a system that would enable management to control all phases of the development, operation and modification of computer programs. Documentation was isolated as the central problem: both one of the largest tasks and the necessary base upon which other functions might be based. In this section we present brief descriptions of additional tasks which ESDP might be extended to perform and for which would use the actual documentation described elsewhere, or the methodology of ESDP for this performance.

The first of these extends documentation from programs and data to program tests, then extends test composition and documentation into general hypothesis testing. The latter is concerned with aiding systems management personnel to study the probable effect of changes in system parameters or logic.

A second major extension is into active computer assistance in programming, itself, rather than solely into the documentation of programs and their designs.

2. Computer Assisted Test Design and Hypothesis Testing. Increasingly, in large programming systems, the programmer is required to provide formal documentation for program testing. Typically, there may be a requirement for each programmer to submit a test plan for his individual program and have it approved by Systems Management before he begins testing. At the conclusion of his tests, he must report on tests actually conducted and the results thereof. As the project moves into the system, or assembly, test phase even more complex test plans are required.

a. Nature of Program Testing

Testing and debugging are complex, ill-defined activities. In a large system it is totally impractical to try each possible input that a program might someday have to process. Since this is so, after any test period there might be a program path or combination of circumstances that has remained untested but would result in an error if executed. One alternative is to systematically test each thread of a program without attempting to try each possible combination of subthreads.

As long as a test designer is going to be selective rather than exhaustive he needs assistance in selecting paths to be tried. Such assistance should be in the form of help in determining which control decisions are independent of each other (to reduce redundant testing), what paths would be followed if input and data base values are given, and what data values are required to reach a given point in a program. In the early

stages, testing and debugging do not differ. We define as the difference that debugging is a process by which a programmer convinces himself that his program "works" and testing is a process by which he formally demonstrates the acceptability of the program to another person.

During debugging, many errors are detected and changes made. Hence, debugging operations are set up to assist the programmer in the rapid detection and correction of errors. During testing this may happen, but the programmer does not expect many errors or, presumably, he would not have entered the test phrase.

In both debugging and testing of individual programs, emphasis is on systematic testing of paths within a program, usually based on artificially created test inputs, sometimes anachronistically called "test decks." In the later stages of program testing--system or assembly testing--whenever large groups of programs are run together, testing is more and more based on test cases which are defined in terms of overall system conditions or situations, rather than in terms of execution sequences. In other words, a system test might be set up on the basis of an externally meaningful test case (e.g., in an air traffic control system, conditions of runway usage, wind, weather, and traffic load may be tested, rather than an explicit test of how programs A, B, and C interrelate).

We have, then, what amounts to a continuum of test situations, ranging from the typical debugging situation of testing a small segment of code by itself, through testing of interactions among programs, to testing a program system's ability to perform properly on an externally-defined condition.

To plan tests at the detailed end of this spectrum the programmer needs such information as:

data items in use

initial values needed for a test

output items computed, switches set, etc.,

and these are usually readily identifiable from a glance at the source coding of a program.

As the programmer begins to test branching and interaction among components of his own program, he is dealing with much larger sets of code and with more complex interactions, particularly with interactions not explicitly recognizable from the text. It is in this area that some of the controversy over the value of on-line debugging arises, proponents arguing that a programmer must spend a large amount of time relearning his own program after each debug run because he cannot retain these complex interactions in his mind for long. Increasingly, the programmer is concerned with such questions as, "What will happen

if the value of X is ___?" "What value of X will cause a transfer to location Y?" "What data items are used to compute the value of X?"

For both individual program testing and for component testing, the programmer must find the answers to such questions as, "Under what circumstances will the program branch from component Y to component Z?" "If input item X = ____, what thread will the program follow?" "In what form is data passed/received between programs A and B?"

Finally, at the highest level of testing, the test designer is concerned with such questions as "What happens when the operator hits the DELETE key after entering a query?" "How will the system react to an overload situation?" "What is the total delay time between the arrival of a given input and its subsequent display on a console?" "What happens if a customer's balance goes to zero?"

In debugging, questions such as these are asked and answered informally. In test planning, they must be asked formally and answered formally, to produce the documentation required and to assure that all threads are tested and that this is done at reasonable cost. The information acquired and produced by ESDP (if expanded and elaborated upon somewhat, but not basically altered) can answer these questions, and the query, retrieval, and report generating facilities of ESDP can be used as the working tool for browsing through this information, answering questions, and compiling draft notes to be used in creating test design documentation.

By extending ESDP slightly we can encompass the preparation of test documentation within ESDP and also provide means for automatic creation of "test decks" or program test data. Preparation of test data is not a difficult conceptual problem but it is a practical, clerical problem of some magnitude which can be alleviated by a well-designed language specific to the function, or, as proposed here, by use of CAINT to elicit requirements and produce data-generating code.

The expansion necessary, in the programming area, would be to increase the amount of detail acquired by program analysis, particularly in classifying the way in which a data label is used in a source code UOP, and somewhat modifying the interrogation to more fully describe branching decisions. The symbolic test data composition program would be similar to a portion of the Lincoln Checker, [7] part of the utility system developed for use in SAGE. In addition to these programming steps, considerable work needs to be done to learn how to make effective use of such facilities and, possibly, how to assess analytically the validity of a test plan, in terms of completeness, redundancy, or efficient use of resources.

b. Hypothesis Testing

Hypothesis testing is an extension of these concepts. In hypothesis testing we assume we have an operating computer system which we wish to change--by modifying equipment, input rate, load, legal input values, or processing to be performed or decision rules to be used. Basically, we wish to answer the question, "What would happen to the existing system if the following change were made?" or "How will the system respond to the following change?" We will, then, be assuming some change in configuration, file content, or program logic, and then tracing through an existing, presumably correctly functioning, program to find out what would happen under the hypothesis that some components have been changed in a specified way. Actually, we would be trying to find out what changes are needed in the other components, given a change in one.

Having this capability would enable a system user, whose background is more in the operational or application area of the program system rather than in programming itself, to understand the workings of the system and to a large extent, to be able to make his own decisions on whether an existing system can respond to a request to adapt to a new requirement, how difficult it would be, etc. In a system of sufficient complexity, this approach may be the only way these questions can be answered, and it might be that a joint team of programmers, system managers, and system users should interrogate the system to determine the change requirements. Thus, hypothesis testing addresses itself to the original problem upon which ESDP was founded--how to control the orderly evolution of a programming system as requirements for its use, or the environment under which it operates, changes.

c. Planning a Test

A test plan for a computer program or program module is, or we believe should be, a plan for a series of program runs which when carried out will demonstrate to a disinterested observer that the program performs according to specifications. A test plan requires both a systematic plan for testing portions of the program and a set of test cases that will force the program to operate as planned. Not all programs are tested in all combinations of paths or conditions. The completeness of a test plan varies with the nature of the program and its use. Often, a test planner, while realizing he is not exhaustively testing a program, has no way of knowing exactly what portions he is testing. Thus, areas left untested are "selected" at random. Areas that are tested may be overtested. The objective of applying ESDP to program test planning is to give the test designer control over those variables.

In order to plan a test, the designer must be able to isolate and describe subsets of the program to be tested in each trial. We expect the ESDP documentation and approach to program organization to be of great help here. There is more than one

way to break down a program but, so long as the various partitions can be described in terms of UOP's, ESDP allows for individual variation. Two major variations are: partitioning a program in terms of what it accomplishes (e.g., adding a new employee record to the personnel file) or in terms of the sequence of execution of UOP's (recall that a single command or program statement can be a UOP). The latter approach to testing, systematically probing possible execution sequences is certainly well adapted to ESDP. The other approach--defining an execution sequence in terms of accomplishment--can be used as the basis for testing, with the program region so defined being translatable into UOP's by use of the ESDP retrieval system. This, in turn, can lead a search from operational terms to UOP's to execution sequences. The ESDP retrieval system is also capable of retrieving and displaying a table of information on control variables for any defined path. This information would become the basis upon which the test designer creates his trial data or test cases.

The generation of test cases, particularly if it is a matter of initializing a set of core-stored variables (as opposed to creating and storing a data set on a disk) is not too difficult but is time-consuming and subject to human error. The technique used in instruction, of having a CAINT program generate PL/I code, can be used to assist test-case generation. The kind of program to be produced would be a relatively simple one that sets a number of data variables to values specified by the test designer. His specification would be entered via a CAINT program which would provide him the information he needs on paths and UOP's, elicit a test specification, and check the validity of test values.

This is one of the almost hidden problems of program testing. Given that a programmer intends to test the path from UOP A to B to C, and he thinks he knows what test values he must use to do so, did he, in fact, compose a correct test case? Using CAINT, each test value can be checked against the specification for the variable, and all test cases cataloqued. After the program has been run, the values of certain variables can be copied and recorded with other test information to completely document the test.

Returning to the mechanics, a test case interrogation program would elicit from the test designer the names of UOP's to test and the test values required. Values for variables would be checked for conformity with specifications. Then, a test case generating program would be produced. This would be a PL/I program to be run just before the test to initialize the stated variables. If ESDP were to maintain the object system program deck, it could also prepare a small program, to be inserted into any test module, that would record the values of stated variables after a test. In this way, both interrogation on input and recording of results would be automatic, saving programmer time, increasing reliability, and increasing credibility of the test process.

The design of a test would probably proceed somewhat along these lines. The test designer has a fairly good idea of the structure of the module he is to test and of the input-to-output transformation it makes. The test designer is not always the programmer, so his familiarity may have to be acquired and he may make heavy use of ESDP's retrieval and instructional facilities in the process. Presumably, he will then make a decision whether to test on the basis of systematic forcing of threads in the program or testing in terms of accomplishment or function, which would require the determination of paths involved.

To force a path--i.e., to get the program to execute a specified set of UOP's in a specified sequence, the tester needs to know what data variables are used in branching decisions, what variables are used to compute control items, and what input or output operations are performed. He compiles, for each path or thread, a set of initial conditions and a list of items that are changing along the path and, of those, which ones he would want recorded as proof of the text. All this information can be recorded with ESDP's facilities and to it should be added some narrative on the purpose or meaning of the test.

d. Test Priorities

It is not necessary that all trials be completed before a program can be used. It is often necessary or at least highly desirable to prove out one portion of a program and use that to help prove out another portion. Since not all paths of all programs will be tested, it is necessary to devise some priority ordering scheme for paths and regions. This priority is not necessarily inherent in the program but may be a function, as well, of the program structure, the test conditions, and the immediate need for the program either operationally or to support other tests.

Criteria for priority ordering of paths and regions should be established by the test designer and thereafter used by the test assistance programs to help him keep track of what he has already done and needs to do next and the relative importance of paths yet to be tested.

An example of the logic that might be used in establishing priorities is this. The designer selects a path in the program which represents the main execution path--the sequence of UOP's to be executed in the most straightforward, error-free situation. If he can get that much tested, he can use his knowledge that it runs correctly to help debug or test other portions and can even allow this program to be used in limited assembly testing while he completes testing of the more obscure paths. Even to test the "main trunk" he may find it desirable to pre-test certain frequently-used subroutines. Thus, these subroutines become first in priority. Next comes the main execution path which establishes that the main program cycles and provides output for use by connected programs. Finally, other

paths than the main one are attacked. These may be further classified according to probability of occurrence, support to debugging and testing, interface with other programs, etc.

At all times during test planning the assistance program keeps track of what trials have been generated, what paths or regions are used, and what test data is used. This information, together with the assistance program's knowledge of the structure of the test program and of the priorities assigned to components of it, enables recommendations to be made to the test designer on what is the most fruitful trial to begin next.

e. Summary

Assistance can be provided to test designers, through ESDP or extensions of the programs specified to date, in the following ways:

(1) Retrieval and display of information on program structure and data usage, used to acquire an understanding of the program and to select paths and regions for trials.

(2) Assistance in tracing execution paths as a function of some criterion, e.g., paths that might be followed if X = 3. While this task might ultimately be mechanized, it would probably be a man-machine function at first.

(3) Acquisition, storage and retrieval of path and region priority information. The object is to allow each designer full freedom to assign priorities (or not use the facilities) as he sees fit, yet make use of the program as easy as possible.

(4) Active assistance by the program in suggesting what paths or regions should next be considered for a test.

(5) Retrieval and display of information relative to data values needed to proceed along a given path, into a region, or to accomplish a stated objective.

(6) Computer assistance in generating test cases by initializing variables to appropriate values.

(7) Computer assistance in recording data values during or after a test, and, in general, in documenting test results.

3. Computer Assisted Programming. Under ESDP we have developed techniques for describing program and data structures, for eliciting information from a programmer about proposed programs and data files and for analyzing programs once written. From these analytic and elicitation programs documentation of program systems is created. Also as part of ESDP, a program has been

produced which converses with a program author and produces an output which is a syntactically correct PL/I program to be used for instructional purposes. These techniques can be combined and generalized to produce a conversational program which can carry on an English-language conversation with a programmer and produce programs of far more general structure than that of the instructional programs generated so far.

The approach would be somewhat the same as now used for documentation. The programmer is first interrogated on his design and, in succeeding interrogations, he enters increasingly more detailed design information. At any time, the programmer may request to be switched into the program composition mode which is really nothing more than a far more detailed design interrogation.

The existence of design information, written by the programmer, provides the ESDP system with information that can be used to search a library of available routines and one of known program models. From either or both these libraries, the program can retrieve information on which to base more detailed interrogations of the programmer. For example, if the design information specifies that a sort is to be performed, the CEL program can retrieve information about sort programs from its library. It detects the relevance of sort programs by a key word analysis of the programmer's responses to documentation questions. With the additional knowledge, it retrieves, it can make further checks against the parameters of existing program modules. If, for example, it discovers that the programmer wishes to sort a table that is stored in core, the interrogation program will try to find such an internal sort program in its library. If it does, little more may be needed than to evaluate a calling sequence. If no program exactly fits, one can be composed either in programming language statements or in terms of smaller modules or macro-instructions. The less prior information that is available, clearly, the less the computer is able to help write the program. Ultimately, if the programmer wishes to compose a program bearing no detectable relationship to any program known to the system, he must write it more or less in the normal manner. Computer assistance can still be used, however. His statements can be syntactically checked as they are entered and documentation, both from interrogation and program analysis, can be built up as the program is composed. This gives the programmer instant review capability, always with fairly complete documentation, on all completed portions of the program.

The greatest value to be derived from this system would come when there is prior information about the type program to be written. If not available as parameter descriptions for an existing program, this information might be largely in the form of <u>object models</u>, or generalized program specifications, to be detailed by a programmer.

The object model is a set of information descriptive of a wide class of object programs, such as sort programs or

instructional programs. The model gives the general structure and limitations and names elements that must be specified for any particular member of the class, or implementation. In the instruction program class, this includes questions, anticipated answers, and branching commands. The Generator program operates through the object model, acquiring from that model both the questions it needs to ask the program designer and the information it needs to analyze his answers.

Use of the object model in conjunction with a generalized Generator (which, recall, is a CEL interrogation program) is analogous to an information retrieval program which makes use of file directories or format tables. By using tables which describe the data files handled by the retrieval system, the retrieval programs can be made quite general. They always refer to the directory to determine how to interpret any file. Hence, it is not necessary to modify a program when changing the structure of a file. It is only necessary to change the directory's description of the file and the program adapts itself to the new structure. Similarly, the Generator will perform the functions specified by the object model so that the Generator need not be modified to change from producing one type program to another. Probably, the more experienced programmers would provide the models and less experienced programmers would add the detail. Early in the development of a large system, the senior designers would contribute specifications in the form of documentation and object models. These models can be successively more detailed as they are passed down the chain of command to less experienced people. The major difference in approach here over what is proposed for documentation purposes is that at the design documentation level, designers supply not only known information but describe what additional information has to be provided in order to proceed to the next level of detail.

To implement this system, a conversational program would be required whose inputs are an object model and a set of user responses entered through a remote terminal. The output would be a computer program in a specified programming language, such as PL/I, ready for compilation and guaranteed to compile. There will be no syntax errors in the object code. Clearly, whether or not there are logical errors depends, as usual, on the designer.

A simplified, preliminary model exists and is at this date partially operational. This is the Instruction Generator previously mentioned (Section I.4) which has as its sole aim the generation of instructional (CAI) programs. Such programs have a far simpler structure than most computer programs. Still, the concept is basically the same, and the programming techniques used to generate object code from English language conversation are directly applicable to the larger, more general problem.

The Generator would enable a user who is not trained in computer programming to write programs with relative ease. Use of the Generator could also increase the effectiveness to

technical personnel who need programs to support their own work. Examples of such people are engineers or scientists, not professional programmers, who nonetheless need data processing services. Other examples are professional computer programmers who must cope with the mysteries of a new operating system in order to accomplish their own tasks. That is, they are users of an operating system in the same sense that an engineer is a user of FORTRAN.

BIBLIOGRAPHY

[1] Meadow, Charles T. and Douglas W. Waugh, "Computer Assisted Interrogation," AFIPS Conference Proceedings 1966 Fall Joint Computer Conference, Spartan Books, Washington, D. C., 1966.

[2] Mills, Harlan D. and Michael Dyer, "Evolutionary Systems for Data Processing," in Proceedings of the Real Time Systems Seminar, IBM Corporation, Houston, Texas, November 2-4, 1966.

[3] _____ System/360 FLOWCHART (360A-SG-22X) Application Description, Form H20-0199, IBM Corp., White Plains, N.Y., 1966.

[4] _____, IBM 7090/94 AUTOFLOW System, User's and Operator's Manual, Applied Data Research, Inc., Washington D.C., February, 1967, under NASA contract no. NAS5-10021.

[5] _____, IBM System/360 Operating System. Queued Telecommunications Access Method, Form C30-2002, IBM Corp., White Plains, N.Y., 1967.

[6] _____, General Information Manual: Selective Dissemination of Information, Brochure E20-8092, IBM Corp., White Plains, N.Y., 1962.

[7] Hill, A., Checker Manual, Doc. No. 6M 4007, MIT Lincoln Laboratory, Lexington, Mass., Dec. 7, 1956.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Center for Exploratory Studies
Internation Business Machines Corporation
Rockville, Maryland 20850

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE
EVOLUTIONARY SYSTEM FOR DATA PROCESSING
SYSTEM DESCRIPTION

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
January 1968

5. AUTHOR(S) *(First name, middle initial, last name)*
Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

6. REPORT DATE
January 1968

7a. TOTAL NO. OF PAGES
55

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO.
F19628-67-C-0254

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
ESD-TR-68-143, Vol. I

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Command Systems Division, Electronic Systems Division, Air Force Systems Command, USAF, L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

ESDP is a proposed system whose purpose is to acquire, store, retrieve, publish and disseminate all documentation, exclusive of graphics, concerned with a large computer programming activity. Documentation is deemed to consist, not only of final or formally published after-the-fact reports, but of working files, design and change notices, informal drafts, management reports -in fact, the entire recordable rationale underlying a programming system. Maximum attention has been concentrated on the means of acquiring and organizing documentation. Two major, complementary approaches are proposed. The first is called Program Analysis and is a process of extracting documentation directly from completed programs. The second is called Computer Assisted Interrogation and is a process of eliciting information directly from the programmer, through on-line communication terminals. The former provides canonical data about the program's structure. The latter provides explanatory material about all aspects of the program, and in the absence of canonical data, may provide tantative structural information as well. The conclusion of the study group is that ESDP is a feasible concept with present-day technology and that it will materially benefit using organizations in the production of programs and in guiding their evolution as requirements change. Its value will be greater for larger organizations, whose internal communications difficulties tend to cause truly gigantic inefficiencies. Its implementation as a support system for such projects would require a significant guantum of investment in order to produce these benefits and is predicated on the use of a computer system dedicated solely to the use of ESDP.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification